BLASを用いた高精度な行列積アルゴリズムの使用メモリ量の削減とその性能について (科学技術計算における理論と応用の新展開)

AUTHOR(S):

尾崎, 克久; 荻田, 武史

CITATION:

尾崎, 克久 ...[et al]. BLASを用いた高精度な行列積アルゴリズムの使用メモリ量の削減とその性能について (科学技術計算における理論と応用の新展開). 数理解析研究所講究録 2012, 1791: 66-75

ISSUE DATE:

2012-04

URL:

http://hdl.handle.net/2433/172838

数理解析研究所講究録
第1791巻 2012年 66-75

BLASを用いた高精度な行列積アルゴリズムの使用メモリ量の削減とその性能について

尾崎 克久 * 荻田 武史 †

* 芝浦工業大学 システム理工学部 数理科学科 (ozaki@sic.shibaura-it.ac.jp)
† 東京女子大学 現代教養学部 数理科学科

概 要

近年，著者らはアルゴリズム中の主計算が行列積という特徴を持つ高精度な行列積アルゴリズムを提案している．最適化されたBLASを有効利用したことにより，内積単位を高精度に計算するアルゴリズムに対して高速化は達成されたが，一方で使用するメモリ量は膨大である．本報告ではこのアルゴリズムの実行に必要なメモリ量を削減するアルゴリズムを提案し，数値実験によりそのパフォーマンスを示す．

Memory Reduced Implementation of Error-Free Transformation of Matrix Multiplication and its Performance

Katsuhisa OZAKI*, Takeshi OGITA†

* Departmarnt of Mathematical Sciences, Shibaura Institute of Technology
† Division of Mathematical Sciences, Tokyo Woman's Christian University

Abstract

Recently, an algorithm for an error-free transformation of matrix multiplication into sum of floating-point matrices has been developed by authors. It is shown that this algorithm is useful for accurate computations. Moreover, computational performance of the algorithm is significantly high. However, as a drawback, this algorithm requires much amount of working memory. In this report, we reduce the amount of working memory without changes of basic flow of the algorithm. Finally, we show the efficiency of the proposed algorithms by numerical experiments.

1 まえがき

本報告では行列積を高精度に計算することに役立つ，行列積のエラーフリー変換という技術について取り扱う．数値計算で代表的に使用されるIEEE 754-2008規格 [1] による浮動小数点演算を利用して行列積を計算した場合，有限精度に起因する丸め誤差が演算毎に発生する可能性がある．この誤差の蓄積により，最悪の場合は計算結果の相対精度が悪いことがあり，1桁も正しい情報を持っていないことさえある．一般的に精度の問題を克服するには，下記の有名なライブラリや手法

- 多倍長精度演算 [11, 12]
- 混合精度演算 [7, 9]
- 内積や総和に対する高精度な計算法 [4, 5, 6, 8]

などを利用することは有力である．本稿ではこれらの手法とは異なり，BLAS (Basic Linear Algebra Subprograms) の Level 3 演算を有効利用した高精度な計算法 [10] を取り扱う．この手法は非常に高速であるが，実行に膨大なメモリ量を必要とする．本稿ではこの先行研究に対して，アルゴリズムの詳細を変えることなく，実装レベルで必要なメモリ量を削減する手法を提案し，そのパフォーマンスを数値実験により示す．

2 高精度な行列積アルゴリズムの使用メモリ量

本章では，提案された高精度計算アルゴリズム [10] を簡単に紹介し，実装に必要なメモリ量を考察したい．$\mathbb{F}$ を IEEE 754-2008 規格が定める浮動小数点数の集合とする．$\mathbf{u}$ を roundoff unit とし，IEEE 754 規格が定める倍精度浮動小数点数であれば $\mathbf{u} = 2^{-53}$ である．$\mathrm{fl}(\cdots)$ は，括弧内の数式を浮動小数点演算（最近点への丸め）で評価することを意味する．ただし，$\mathrm{fl}(\cdots)$ の評価ではオーバーフローやアンダーフローは発生しないと仮定する．行列を $A = (a_{ij}) \in \mathbb{F}^{m\times n}$ と $B = (b_{ij}) \in \mathbb{F}^{n\times p}$ とし，$n \ll \mathbf{u}^{-1}$ を仮定し，定数 β を

$$\beta = \left\lceil \frac{\log_2 n - \log_2 \mathbf{u}}{2} \right\rceil \tag{1}$$

とする．またベクトル $\sigma \in \mathbb{F}^m$ と $\tau \in \mathbb{F}^p$ を

$$\sigma_i^{(1)} = 2^\beta \cdot 2^{v_i^{(1)}}, \quad \tau_j^{(1)} = 2^\beta \cdot 2^{w_j^{(1)}}$$

とする．ここでベクトル $v^{(1)} \in \mathbb{F}^m$ と $w^{(1)} \in \mathbb{F}^p$ は

$$v_i^{(1)} = \lceil \log_2 \max_{1\le j\le n} |a_{ij}| \rceil, \quad w_j^{(1)} = \lceil \log_2 \max_{1\le i\le n} |b_{ij}| \rceil \tag{2}$$

と定義される．$e = (1, 1, \ldots, 1)^T$ とし，行列 A と B に対して

$$\begin{aligned} A^{(1)} &= \mathrm{fl}\left((A + \sigma^{(1)} \cdot e^T) - \sigma^{(1)} \cdot e^T \right), \qquad \underline{A}^{(2)} = \mathrm{fl}\left(A - A^{(1)} \right), \\ B^{(1)} &= \mathrm{fl}\left((B + e \cdot (\tau^{(1)})^T) - e \cdot (\tau^{(1)})^T \right), \qquad \underline{B}^{(2)} = \mathrm{fl}\left(B - B^{(1)} \right) \end{aligned} \tag{3}$$

を計算する．現在までに

$$A = A^{(1)} + \underline{A}^{(2)}, \quad B = B^{(1)} + \underline{B}^{(2)}$$

という関係が成立する．次に $\sigma^{(2)}$ と $\tau^{(2)}$ を $\underline{A}^{(2)}$ と $\underline{B}^{(2)}$ から

$$\sigma_i^{(2)} = 2^\beta \cdot 2^{v_i^{(2)}}, \quad \tau_j^{(2)} = 2^\beta \cdot 2^{w_j^{(2)}}$$

と定義し，$v^{(2)}$ と $w^{(2)}$ は

$$v_i^{(2)} = \lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(2)}| \rceil, \quad w_j^{(2)} = \lceil \log_2 \max_{1\le i\le n} |\underline{b}_{ij}^{(2)}| \rceil$$

とする．$\underline{A}^{(2)}$ と $\underline{B}^{(2)}$ に対して

$$A^{(2)} = \mathrm{fl}\left((\underline{A}^{(2)} + \sigma^{(2)} \cdot e^T) - \sigma^{(2)} \cdot e^T\right), \qquad \underline{A}^{(3)} = \mathrm{fl}\left(\underline{A}^{(2)} - A^{(2)}\right),$$
$$B^{(2)} = \mathrm{fl}\left((\underline{B}^{(2)} + e \cdot (\tau^{(2)})^T) - e \cdot (\tau^{(2)})^T\right), \qquad \underline{B}^{(3)} = \mathrm{fl}\left(\underline{B}^{(2)} - B^{(2)}\right)$$

を計算する．現在までに生成された行列に対して下記が成立している．

$$\underline{A}^{(2)} = A^{(2)} + \underline{A}^{(3)}, \qquad A = A^{(1)} + A^{(2)} + \underline{A}^{(3)},$$
$$\underline{B}^{(2)} = B^{(2)} + \underline{B}^{(3)}, \qquad B = B^{(1)} + B^{(2)} + \underline{B}^{(3)}.$$

今後，一般的にベクトル $\sigma^{(k)} \in \mathbb{F}^m$ と $\tau^{(k)} \in \mathbb{F}^p$ を

$$\sigma_i^{(k)} = 2^\beta \cdot 2^{v_i^{(k)}}, \quad \tau_j^{(k)} = 2^\beta \cdot 2^{w_j^{(k)}} \tag{4}$$

として，ベクトル $v^{(k)} \in \mathbb{F}^m$ と $w^{(k)} \in \mathbb{F}^p$ を

$$v_i^{(k)} = \lceil \log_2 \max_{1\le j\le n} |\underline{a}_{ij}^{(k)}| \rceil, \quad w_j^{(k)} = \lceil \log_2 \max_{1\le i\le n} |\underline{b}_{ij}^{(k)}| \rceil$$

とする．$A^{(k)}$, $\underline{A}^{(k+1)}$, $B^{(k)}$, $\underline{B}^{(k+1)}$ は，$\underline{A}^{(k)}$ と $\underline{B}^{(k)}$ からそれぞれ

$$A^{(k)} = \mathrm{fl}\left((\underline{A}^{(k)} + \sigma^{(k)} \cdot e^T) - \sigma^{(k)} \cdot e^T\right), \qquad \underline{A}^{(k+1)} = \mathrm{fl}\left(\underline{A}^{(k)} - A^{(k)}\right),$$
$$B^{(k)} = \mathrm{fl}\left((\underline{B}^{(k)} + e \cdot (\tau^{(k)})^T) - e \cdot (\tau^{(k)})^T\right), \qquad \underline{B}^{(k+1)} = \mathrm{fl}\left(\underline{B}^{(k)} - B^{(k)}\right) \tag{5}$$

と得られる．$A = \underline{A}^{(1)}$, $B = \underline{B}^{(1)}$ と見なし，(4) と (5) の計算を $k = 1, 2, \ldots$ と順に実行すると，ある $n_A, n_B \in \mathbb{N}$ が存在し，

$$A = \sum_{r=1}^{n_A} A^{(r)}, \quad B = \sum_{s=1}^{n_B} B^{(s)}, \quad \underline{A}^{(n_A+1)} = O_{mn}, \quad \underline{B}^{(n_B+1)} = O_{np} \tag{6}$$

を満たす．ここで O_{mn} は m 行 n 列の零行列である．以上のように生成された行列に対しては

$$A^{(i)}B^{(j)} = \mathrm{fl}(A^{(i)}B^{(j)}), \quad 1 \le i \le n_A, \quad 1 \le j \le n_B$$

が成立するために，

$$AB = \sum_{k=1}^{n_A n_B} C^{(k)}, \quad C^{(1)} = \mathrm{fl}(A^{(1)}B^{(1)}), \quad \ldots, C^{(n_A n_B)} = \mathrm{fl}(A^{(n_A)}B^{(n_B)})$$

のように，行列積は浮動小数点演算のみを用いて行列の総和に無誤差で変換できる．この総和に対して，高精度な総和のアルゴリズム [5, 6] を適用した結果をそれぞれ $R, S \in \mathbb{F}^{m\times p}$ とすれば，

$$|R - AB| \le 2\mathbf{u}|AB|, \quad |S - AB| \le \mathbf{u}|AB| \tag{7}$$

を満たす．ここで，行列に対する不等式は，成分ごとに不等式がすべて成立していることを意味する．(7) から，提案手法による計算結果の精度が保証されていることがわかる．同様に計算結果の精度を保証するために，行列積の計算に現れる内積単位に [5, 6] を応用することも可能であるが，提案手法は行列積を計算する高速な関数の恩恵を大いに受けることができる．よって，行列の成分間に絶対値の差があまりないとき，すなわち n_A や n_B が大きくない場合には，内積単位の計算を高精度に計算するよりも提案手法が高速であることを著者らは確認をしている．ここで，行列の積から総和に変換するアルゴリズムを $C' = \texttt{accmul}(A, B)$ と表記する ($C' = \sum_{i=1}^{n_A n_B} C^{(i)}$ を意味する)．

ここから，このアルゴリズムの実行に必要なメモリ量を考える．議論を簡単にするために，A と B をそれぞれ n 行 n 列の正方行列とし，この正方行列を格納するのに必要なメモリ量を μ ($8n^2$ バイト) とすると

- A を n_A 個の行列に分解すること ($n_A\mu$)
- B を n_B 個の行列に分解すること ($n_B\mu$)
- $A^{(i)}B^{(j)}$ を格納すること ($n_A n_B \mu$)

が必要であり，アルゴリズムの実行には

$$(n_A + n_B + n_A n_B)\mu \tag{8}$$

というメモリ量が必要である[1]．ここでは，入力される行列 A, B と AB の計算結果を保存するためのメモリ量を含んでいない．この手法は，ベクトルの内積単位で高精度計算をする方法に比べれば非常に膨大なメモリ量を必要とするため，次章に実装レベルで使用するメモリ量を削減する方法を紹介する．

3 メモリ量の削減とそのパフォーマンス

本章では使用するメモリ量を削減する方式を 2 つ，またそれらを組み合わせた方式を紹介し，数値実験結果によってパフォーマンスの変化を確認する．

3.1 部分的に実装

行列積の性質により，$C = AB$ の C は部分的に求めることができる．MATLAB に用いられる表記法を用いると，例えば n が偶数である場合に $C(1:n/2, 1:n/2)$ を求める場合は $A(1:n/2, :) * B(:, 1:n/2)$ を計算すればよい (図 1 を参照)．ここで，$1:n/2$ と記載した場合は，1 から $n/2$ までの範囲であることを意味し，記号 : のみの場合はすべての要素を表すことにするため，$1:n$ と同じ意味である．

[1] 行列を分割するときに行列 $\sigma^{(k)} \cdot e^T$ を使用しているが，MATLAB で実装するときには行列の分割が終了したと同時にメモリ領域を解放し，$A^{(1)}B^{(1)}$ などを格納していくためにこの $\sigma^{(k)} \cdot e^T$ に必要なメモリ量は無視できる．さらに，C 言語等で実装する場合には行列として $\sigma^{(k)} \cdot e^T$ を用意する必要はない．

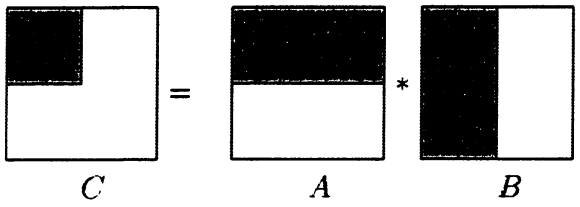

図 1: 分割行列積のイメージ．C の左上のブロックを部分的に計算したい場合には，A と B の半分の要素の情報のみが必要である．

n が k で割り切れると仮定し，C を縦横に k 分割して順次求めると

$$
\begin{aligned}
&d = n/k;\\
&\textbf{for } i = 1:k\\
&\quad \textbf{for } j = 1:k\\
&\qquad C((i-1)d+1:i*d,(j-1)d+1:j*d) =\\
&\qquad\quad \textbf{accmul}(A((i-1)d+1:i*d,:) * B(:,(j-1)d+1:j*d));\\
&\quad \textbf{end}\\
&\textbf{end}
\end{aligned}
$$

という流れである．accmul で使用されるワーキングメモリは，入力の行列のサイズが小さくなるために少なくなる．ただし，accmul が部分的に複数回実行され，行列の同じ要素が複数回分割されるために計算量は増加する．さらに，行列のサイズが小さくなることより，BLAS の行列積関数のパフォーマンスの低下が見込まれる．ただし，行列積に必要な $\mathcal{O}(n^3)$ 回の浮動小数点演算に対し，行列を分割するためには $\mathcal{O}(n^2)$ の浮動小数点演算で良いため，n がある程度大きい場合には計算量の増加は無視でき，BLAS の行列積のパフォーマンスも大きくは低下しない．このように，分割して実行した場合に必要なメモリ量は

$$(n_A + n_B)\mu/k + n_A n_B \mu/k^2$$

となり，(8) に比べて必要となるメモリ量を削減できていることがわかる．例えば $n_A = n_B = k = 4$ のときには，ワーキングメモリは行列 3 つ分を保存する量で良いことがわかる．

ここで，数値実験結果を示す．数値実験には Intel Xeon 5550 (2.66GHz) を 2 つ，合計 8 コアを搭載した CPU と MATLAB2011a を使用した．行列 A と B を MATLAB の randn(n) を用いて生成し，スレッド数と分割数 k を変えたときの計算時間（秒）を行列のサイズ n ごとに表 1 から表 4 に示した．表に示されている数値は，行列積から総和への変換に要した時間を表している．

表 1 から表 4 より，行列のサイズが小さい，かつ分割数やスレッド数が多いときには，計算時間は大きく増加していることがわかる．特に，表 2 にあるスレッド数が 8 でかつ $k = 5$ の場合は，計算時間は 2 倍以上になっている．MATLAB がインタプリタであることからオーバヘッドの問題があり，さらに MATLAB 上の行列・ベクトル積の計算がシングルスレッドで行われていることも原因である．よって $\mathcal{O}(n^2)$ の計算量の増加はここでは無視できない結果となった．行列のサイズがある程度大きいときには，計算時間の

表 1: 計算時間 (秒) の比較 ($n = 1200$).

分割数 (k) / スレッド数	1	2	4	8
1	5.22	2.85	1.72	1.13
2	5.70	3.29	2.17	1.61
3	6.06	3.60	2.50	1.93
4	6.35	3.93	2.70	2.12
5	6.50	4.06	2.82	2.19

表 2: 計算時間 (秒) の比較 ($n = 2400$).

分割数 (k) / スレッド数	1	2	4	8
1	39.7	20.6	11.6	6.91
2	41.5	22.5	13.5	8.84
3	43.1	24.2	15.2	10.5
4	44.7	25.9	16.9	12.2
5	46.4	27.6	18.5	14.0

増加は抑えられていることがわかる．表 4 にあるスレッド数が 1 で $k = 5$ の場合は，計算時間は 1%も増えていない．また，分割を用いる手法はメモリの削減以外にも効果があり，例えば A 全体としては $A = \sum_{k=1}^{5} A^{(k)}$ と分割されている場合でも，部分行列に関しては $A(1 : n/2, :) = \sum_{k=1}^{4} A^{(k)}(1 : n/2, :)$ となることもあり，A の一部分を対象にした場合，計算量が削減される可能性がある．

3.2 分割行列の上書き

提案手法は，$A^{(1)}, A^{(2)}, \ldots$ と逐次的に求める手法ではあるが，$A^{(3)}$ を求めるときには，$A^{(1)}, A^{(2)}$ が必要というわけではない．よって $A^{(1)}$ を求め，$B^{(1)}$ を求めて $A^{(1)}B^{(1)}$ を計算する．次に $B^{(1)}$ を上書きして $B^{(2)}$ を求め，$A^{(1)}B^{(2)}$ を計算する．このように，$A^{(1)}, \ldots, A^{(r)}$ と $B^{(1)}, \ldots, B^{(s)}$ をすべて同時に保持をしていなくてもすべての (i, j) の組に対して $A^{(i)}B^{(j)}$ を計算することは可能である．

表 3: 計算時間 (秒) の比較 ($n = 4800$).

分割数 (k) / スレッド数	1	2	4	8
1	305	155	81.9	48.2
2	310	160	87.5	50.6
3	316	166	93.1	55.2
4	321	168	96.3	59.5
5	325	173	100	64.8

表 4: 計算時間 (秒) の比較 ($n = 9600$).

分割数 (k) / スレッド数	1	2	4	8
1	3809	1970	1031	615
2	3851	1986	1064	639
3	3901	2030	1104	649
4	3933	2046	1126	667
5	3828	2013	1139	685

このアルゴリズムを疑似的に書くと

```
for i = 1 : n_A
  A^(i)を計算する (A^(i-1)があれば上書きをする)
  for j = 1 : n_B
    B^(j)を計算する (B^(j-1)があれば上書きをする)
    乗算 A^(i)B^(j)を計算する
  end
end
```

となる．以上のアルゴリズムでは，$A^{(i)}, \underline{A}^{(i+1)}, B^{(j)}, \underline{B}^{(j+1)}$ のみを保持していればよい．$B^{(1)}$ から $B^{(r)}$ までは n_B 回生成されることになり，計算量は増加する．この手法を使用した場合，必要なメモリ量は

$$4\mu + n_A n_B \mu$$

となる．

表 5 は，前節と同じ計算機環境と行列において，数値実験を行った結果であり，表中の数値は，表 1 から表 4 までの分割数 1 のときに対する計算時間の増加（比）を表す．行列のサイズに関係なく，計算時間の増加を抑えられていることがわかる．

表 5: 計算時間の増加 (比).

行列のサイズ / スレッド数	1	2	4	8
1200	1.05	1.09	1.15	1.28
2400	1.02	1.06	1.12	1.21
4800	1.01	1.03	1.05	1.16
9600	1.00	1.00	1.01	1.06

表 6: 計算時間の増加 (比) ($n = 1200$).

分割数 / スレッド数	1	2	4	8
1	1.05	1.11	1.14	1.29
2	1.18	1.33	1.51	1.89
3	1.27	1.48	1.77	2.22
4	1.39	1.65	1.97	2.75
5	1.46	1.76	2.22	3.14

3.3 複合的な実装

本節では，前節で紹介した 2 つの方法を組み合わせた方法を考える．3.1 で紹介した accmul を分割行列に対して実行するが，その accmul の中では 3.2 で提案したように，生成した行列を上書きしながら計算を行う手法である．3.1 節中の分割数を k とした場合，この手法の実行に必要なメモリ量は

$$4\mu/k + n_A n_B \mu/k^2$$

となる．3.1 節で用いた計算機環境と行列を用いて数値実験を行い，スレッド数と分割数 k を変えたときの計算時間（秒）を行列のサイズ n ごとに表 6 から表 9 に示した．表中の数値は，表 1 から表 4 までの分割数 1 のときに対する計算時間の増加（比）を表す．

表 6 の結果より，行列のサイズが小さい，かつスレッド数が多いときには計算時間が 2 倍以上になることもある．逆に，行列のサイズがある程度大きいときには計算時間の増加を抑えられているため，必要なメモリ量を削減しながらパフォーマンスを維持していると言える．

4 おわりに

本報告では，文献 [10] で提案されている行列積のエラーフリー変換に必要なワーキングメモリの量を削減する方式を提案した．各アルゴリズムに必要なメモリ量は表 10 のとおりである．行列のサイズがある程度大きいときには，先行研究と同じパフォーマンスを保ったまま必要なメモリ量を削減できたことを数値実験により確認した．一般には，サイズが

表 7: 計算時間の増加 (比) ($n = 2400$).

分割数 / スレッド数	1	2	4	8
1	1.02	1.07	1.11	1.21
2	1.09	1.19	1.31	1.65
3	1.15	1.28	1.50	1.91
4	1.20	1.40	1.68	2.22
5	1.26	1.50	1.87	2.56

表 8: 計算時間の増加 (比) ($n = 4800$).

分割数 / スレッド数	1	2	4	8
1	1.01	1.02	1.04	1.08
2	1.05	1.09	1.19	1.27
3	1.08	1.16	1.31	1.46
4	1.11	1.21	1.40	1.66
5	1.15	1.27	1.52	1.84

大きい行列を扱うときにメモリ不足になるために，行列のサイズが大きいときにメモリを削減でき，かつパフォーマンスを維持していることは有益な結果といえる．今回のコードはMATLAB 上で実験しているが，C 言語等で実装した場合にはサイズの小さい行列に対して，さらなるパフォーマンスの向上が見込める．

参考文献

[1] *IEEE Standard for Floating-Point Arithmetic*, Std 754–2008, 2008.

[2] N.J. Higham: Accuracy and Stability of Numerical Algorithms, second edition, SIAM Publications, Philadelphia, 2002.

[3] T. J. Dekker: A floating-point technique for extending the available precision, Numer. Math., 18, 224–242 (1971).

[4] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product, SIAM J. Sci. Comput., 26, 1955–1988 (2005).

[5] S. M. Rump, T. Ogita, S. Oishi: Accurate Floating-Point Summation Part I: Faithful Rounding, SIAM J. Sci. Comput., 31:1, 189-224 (2008).

表 9: 計算時間の増加 (比) ($n = 9600$).

分割数 / スレッド数	1	2	4	8
1	1.01	1.00	1.01	1.06
2	1.02	1.03	1.07	1.22
3	1.03	1.03	1.13	1.26
4	1.04	1.07	1.17	1.27
5	1.06	1.10	1.23	1.31

表 10: 必要なメモリ量.

提案手法	必要なメモリ量
従来法 [10]	$(n_A + n_B + n_A n_B)\mu$
提案手法 (3.1 節)	$(n_A + n_B)\mu/k + n_A n_B \mu/k^2$
提案手法 (3.2 節)	$4\mu + n_A n_B \mu$
提案手法 (3.3 節)	$4\mu/k + n_A n_B \mu/k^2$

[6] S. M. Rump, T. Ogita, S. Oishi: Accurate Floating-Point Summation Part II: Sign, K-fold Faithful and Rounding to Nearest, SIAM J. Sci. Comput., 31:2, 1269-1302 (2008).

[7] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, D. Yoo, Design, Implementation and Testing of Extended and Mixed Precision BLAS, ACM Transactions on Mathematical Software, 28:2, 152-205 (2002).

[8] J. Demmel, Y. Hida: Accurate and Efficient Floating Point Summation, SIAM J. Sci. Comput., 25:4, 1214–1248 (2003).

[9] D. Bailey: A Fortran-90 Based Multiprecision System: ACM Transactions on Mathematical Software, 21:4, 379-387 (1995).

[10] K. Ozaki, T. Ogita, S. Oishi, S. M. Rump: Error-Free Transformation of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and its Applications, Numerical Algorithms, 59:1, pp. 95-118 (2012).

[11] The MPFR Library: `http://www.mpfr.org/`

[12] exflib - extend precision floating-point arithmetic library: `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/exflib-index.html`